Chapitre 3 : types de données

Notion de type de données

Types primitifs

Le type `String`

Notion de type de données

- Tous les objets ont un **type**.
- Un type définit l'ensemble des valeurs que peut prendre un objet informatique ainsi que les opérations permises sur ces valeurs.

Exemple de type :

Ensemble de valeurs : $\mathbb{N}$

+

Ensemble d'opérations sur ces valeurs `(+,-,*,/,==,...)`

- Notion d'objet informatique
  - Toute entité à laquelle un type est associé (variable, constante, fonction, procédure, exception, package, instance, ...) est un objet informatique.

Intérêt du concept de type

- Comment fournir à la machine les informations nécessaires pour qu'elle soit en mesure de vérifier que l'utilisation des objets d'un programme est bien conforme à l'intention préalable du programmeur ?

- Lorsque nous déclarons un objet (variable, constante, fonction, ...), nous spécifions les contraintes de son utilisation, l'ensemble des valeurs qui pourront lui être associées et ainsi définir le cadre légitime de son utilisation.

- L'utilisation de cet objet sera précédée d'une vérification effectuée par le compilateur. La valeur qu'il prend appartient-elle à cet ensemble ? Son utilisation est-elle conforme aux contraintes spécifiées ?

- Le concept de type permet d'établir un lien sémantique entre la déclaration et l'utilisation d'un objet.

Type de données primitifs (1/2)

- Chaque type possède un **nom**, identificateur construit selon les règles lexicales du langage

- On distingue les types numériques
  - types entiers : `byte,short,int,long`
  - types réels : `float,double`

- le type booléen : `boolean`

- le type caractère : `char`

Représentation interne des réels

En notation scientifique un nombre réel x est exprimé sous la forme : $\pm a * 10^n$
où a, la mantisse est un nombre compris entre 1 et 10 (exclu),
n l'exposant de x et $\pm$ le signe de x.

	nombre de bits	signe	exposant	mantisse	plage des valeurs représentées
simple précision	32	1	8	23	-3.4E38→3.4E38 6 ou 7 décimaux significatifs
double précision	64	1	11	52	-1.7E308→1.7E308 14 ou 15 décimaux significatifs

Représentation interne des réels (1/2)

S	E exposant sur 8 bits	mantisse M sur 23 bits

norme IEEE754

S : 0 => nombre positif, 1=> négatif

E : exprimé en puissance de 2
sa valeur entière est interprétée comme E-127

M : le premier chiffre significatif étant 1, il n'est pas représenté
en réalité, valeur réelle en base 2 => $(1.M)_2$
exception lorsque E=0, la valeur réelle est alors 0.M

Représentation interne des réels (2/2)

Exemple : représentation interne de 9.75 en simple précision

1- Signe S = 0 => nombre positif

2- Calcul de la mantisse

partie entière en binaire :

```
9|2
1 4|2
  0 2|2
    0 1|2
      1 0 => 1001
```

partie décimale en binaire :

```
0.75*2=1,5 => 1
0,5*2=1 => 1
représentation de 0,75 => 11
```

mantisse :

```
1001,11
1,00111 10_2^11
1,00111 E
```

3- Exposant

```
exposant = E-127
E= exposant + 127 = 3+127
E= 00000011+01111111 = 10000010
```

4- Représentation finale

```
0 10000010 00111000000000000000000
```

le 1 n'est pas représenté (non significatif)

Représentation interne des entiers

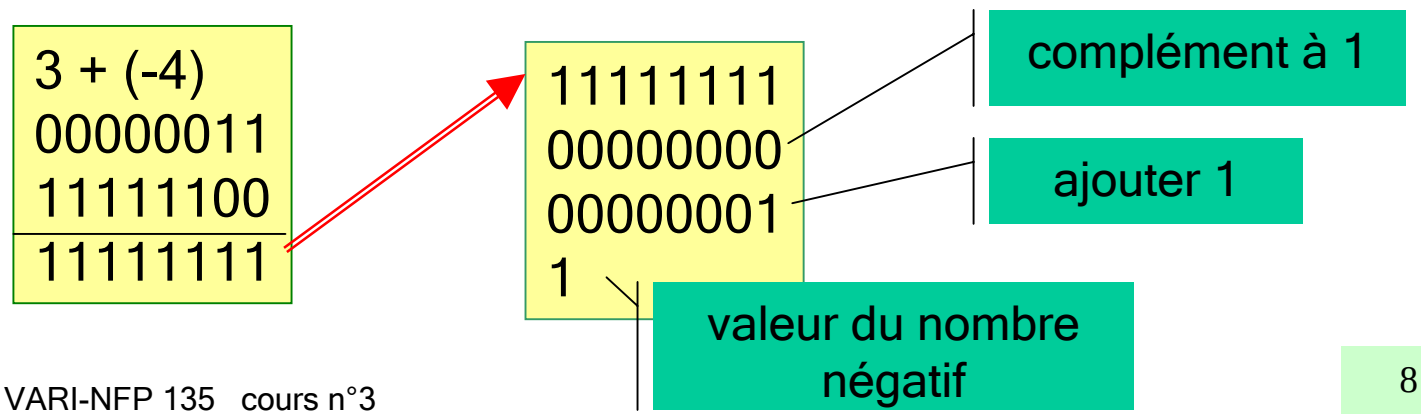
représentation des entiers => représentation binaire

entiers naturels (N)

entiers relatifs (Z) les négatifs en complément à 2

	4	→ 00000100
complément à 2 =>	complément à 1	→ 11111011
	on ajoute 1 → -4	→ 11111100

Exemple



Types primitifs

Nom du type	Ensemble des valeurs	Nombre d'octets
`byte`	[-128..127] ($[-2^7..2^7-1]$)	1
`short`	[-32768..32767] ($[-2^{15}..2^{15}-1]$)	2
`int`	$[-2^{31}..2^{31}-1]$	4
`long`	$[-2^{63}..2^{63}-1]$	8
`float`	réels simple précision (7 chiffres significatifs)	4
`double`	réels double précision (15 chiffres significatifs)	8
`boolean`	`true false`	1
`char`	un caractère	2

Valeurs littérales numériques

valeur	2	2.0	2.0f ou 2.0F	2.0d ou 2.0D	2l ou 2L	2f ou 2F	2d ou 2D
type	`int`	`double`	`float`	`double`	`long`	`float`	`double`

Un entier suivi de `f (F)` devient un `float`
Un entier suivi de `d (D)` devient un `double`

La représentation physique de la valeur dépend du type associé

Opérateurs arithmétiques

opérateurs sur les types entiers :

- `byte,short,int,long`
- `+, -, *, /, %`
- `9/4 = 2`
- `9%4 = 1`

opérateurs sur les types réels

- `+,-,*,/`
- `9/4 = 2,25`

opérateurs relationnels

- `<, >, <=, >=, ==, !=`
- exemples : `5 > x, a == 'b', (a != 0) && (a <= 100)`

Précédence des opérateurs (1/2)

Les expressions composées de plusieurs opérateurs sont évaluées de la gauche vers la droite, selon des *règles de précédence* indiquant la priorité des opérateurs les uns par rapport aux autres.

`2 + 3 * 4` --> équivaut à `2 +(3*4)`

▫▫▫ s'évalue en `14` et non pas en `20`

`2 + 3 > 7` --> s'évalue en `false`

`2 + (3 > 7)` --> produit une erreur

Précédence des opérateurs (2/2)

De la plus haute à la plus basse priorité

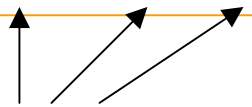
```
opérateurs unaires +,-
opérateur booléen !
* / %
+ -
< <= > >=
== !=
^
&&
||
```

L'opérateur conditionnel ternaire ?:

```
z = (x>y)? x : y;
```

≡

```
if (x>y) z = x;
else z = y;
```



3 opérandes

Si la première opérande vaut `true`, l'expression a pour valeur celle de `x` sinon elle a pour valeur celle de `y`

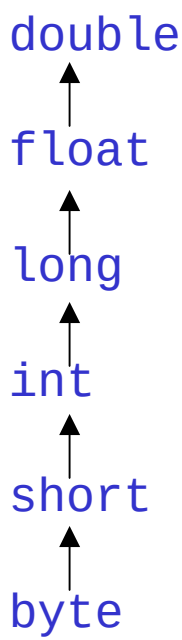
```
int x = ......;
System.out.println(
      (x%2==0)? x+" est pair" : x+" est impair");
```

Opérateurs et expressions

- un opérateur permet de réaliser des calculs sur une ou plusieurs valeurs appelées opérandes.
- il existe des opérateurs unaires (une seule opérande), binaires et ternaire
  - `!true, 9*X`
- les expressions sont des constructions décrivant des opérations à l'aide d'opérateurs et d'opérandes
- les opérandes d'une expression peuvent être des constantes, des variables valuées ou des appels de fonction
  - `(34 + Math.min(X,Y))` où `X` et `Y` sont les arguments valués de la fonction `Math.min`
  - `2>5` est une expression booléenne dont le résultat est `false`
  - `!(2>5)` est une expression booléenne lue :
    `non 2 plus grand que 5`
- une expression correspond toujours à la valeur qu'elle calcule

Conversions implicites entre types numériques

- **cas n°1** : les opérandes sont de même type => le résultat est du même type
- **cas n°2** : pour que l'opération soit réalisée, il faut que toutes les opérandes soient de même type => nécessité de convertir certaines opérandes.
  La conversion se fait vers le type le plus large selon la hiérarchie :



Exemples :

```
 6.2-4 => 6.2-4.0
5.4f+55.8 => 5.4+55.8
```

Remarque :
aucune perte d'information

Conversions explicites entre types numériques

- Une conversion explicite est appelée cast en Java
- Syntaxe : `(type_cible)val`
  - `type_cible` est le type vers lequel la conversion est effectuée
  - `val` la valeur à convertir
- Exemples
  - `(int)64.89` => `64` on passe d'une représentation sur 8 octets vers une à 4 octets avec changement de codage de la valeur
  - `(float)64.89` => on passe d'une représentation sur 8 octets vers une à 4 octets sans changement de codage de la valeur

Remarque :
avec perte d'information

Le type `Boolean`

- ensemble des valeurs du type : `true, false`

- ensemble des opérateurs : `&&,||,!`
  - `&&` conjonction, se dit "et"
  - `^` disjonction exclusif
  - `||` disjonction, se dit "ou"
  - `!` négation, se dit "non"

- exemples
  - `true && false => false`
  - `!x ||(b && c)`
  - `(2>4)^(5>3) => true`
  - `('a'<'y' && 'b'>'x') => false`
  - `((X%400==0) || (X%4==0 && X%100!=0)) => true`

```
// si X représente une année bissextile
```

Tables de vérité

p	q	p && q
v	v	v
v	f	f
f	v	f
f	f	f

p	q	p || q
v	v	v
v	f	v
f	v	v
f	f	f

p	q	p ^ q
v	v	f
v	f	v
f	v	v
f	f	f

Simplification des expressions booléennes (1/2)

On peut utiliser les lois d'équivalences suivantes :

anihilation

```
p || true  ≡ true
p && false ≡ false
```

loi de Morgan

```
!(p || q) ≡ !p && !q
!(p && q) ≡ !p || !q
```

élément neutre

```
p || false ≡ p
p && true  ≡ p
```

simplification

```
p || (p && q) ≡ p
p && (p || q) ≡ p
```

idempotence

```
p || p  ≡ p
p && p  ≡ p
!(!p)   ≡ p
```

élimination des négations

```
p || (!p && q) ≡ p || q
p && (!p || q) ≡ p && q
```

distribution

```
p || (q && r) ≡ (p || q) && (p || r)
p && (q || r) ≡ (p && q) || (p && r)
```

Simplification des expressions booléennes (2/2)

```
(a<b) || ((a>=b) && (c==d))
```

est de la forme

```
p || (!p && q)
```

application de la distribution

```
(p || !p) && (p || q)
```

anihilation

```
true && (p || q)
```

élément neutre

```
(p || q)
```

c'est à dire

```
(a<b)||(c==d)
```

Le type primitif `char`

- Codage sur 16 bits
- Permet l'échange, le traitement et l'affichage de texte dans différentes langues (*www.unicode.org*)
- L'unicode utilise 2 octets précédés par \u exprimés par 4 chiffres hexadécimaux (de `'\u0000'` à `'\uFFFF'`)
- Permet de représenter 65.536 caractères
- Exemples :

  `'A' ≡ '\u0041';`

  Unicode inclut les 128 caractères ASCII (de `'\u0000'` à `'\u007F'`)

  `'\t'` → tabulation
  `'\n'` → nouvelle ligne
  `'\f'` → nouvelle page

  `'\u03b1'` → α
  `'\u03b2'` → β
  `'\u03b3'` → γ
  `'\u20AC'` → €

Le type `String`

Le type `String` n'est pas un type primitif. Ses valeurs sont des objets représentant des chaînes de caractères.

Les valeurs du type `String` s'écrivent entre quotes:

```
"hello"  "123" "__XX_"
"mon nom est : \"Martin\"!"
```

Opérateur de concaténation : `+`

La chaîne : `"hello"+" world"` est identique à `"hello world"`

Utilisation du type `String`

Syntaxe

```
<chaine>.<ident_méthode>(<liste_param>)
```

`<chaine>` est une valeur du type `String`

`<ident_méthode>` est un nom de méthode applicable sur les valeurs du type `String`

`<liste_param>` le ou les paramètres séparés par une virgule

Exemple

```
int i = "toto".length(); // i ←4
```

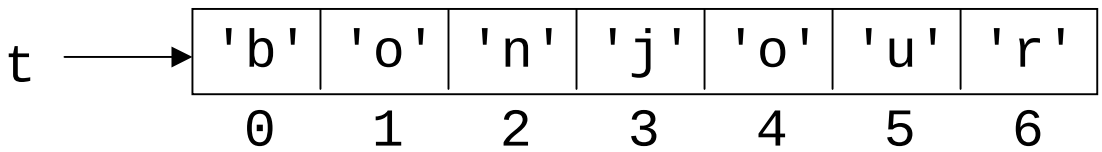
Quelques méthodes du type `String`

`char charAt( int n )` retourne le n-ième caractère de la chaîne

```
char c = "bonjour".charAt(0); // c←'b'
```

`char[] toCharArray()` transforme une chaîne en un tableau de caractères

```
char[] t ="bonjour".toCharArray();
```



`int compareTo(String s2)` comparaison d'une chaîne `s1` avec une chaîne `s2` selon l'ordre lexicographique.

`s1<s2` retourne un entier < 0

`s1=s2` retourne un entier = 0

`s1>s2` retourne un entier > 0

`String toLowerCase(), String toUpperCase()` pour transformer les caractères en majuscules ou en minuscules

Egalité entre chaînes

On distingue 2 sortes d'égalité : l'égalité de contenant (d'objets) et l'égalité de contenu

Les opérateurs : `==` et `!=` testent si deux noms de variables différents désignent le même contenant

```
String s1 = "SOS";
String s2 = s1;
if( s1==s2) System.out.println( s1+" = "+s2);
else System.out.println( s1+" ≠ "+s2);
```

L'opérateur de comparaison des contenus est la méthode `equals(String s2)`

```
String s1 = "SOS";
String s2 = "OSS";
if( s1.equals(s2)) System.out.println( s1+" = "+s2);
else System.out.println( s1+" ≠ "+s2);
```

Conversion vers les types primitifs

String → int

```
String s = "123";
int x = Integer.parseInt(s);  // x←123
```

String → double

```
String s = "123";
double x = Double.parseDouble(s);  // x←123.
```

String → boolean

```
String s = "true";
boolean x = Boolean.parseBoolean(s);  // x←true
String s = "toto";
boolean x = Boolean.parseBoolean(s);  // x←false
```

Idem pour : `String → float`, `String → short`, etc …

Conversion vers le type `String`

int → String

```
int i = 15;
String s = ""+15; // s←"15"
```

double → String

```
double i = 15.23;
String s = ""+15.23; // s←"15.23"
```

boolean → String

```
boolean i = true;
String s = ""+true; // s←"true"
```